Storing Music Metadata

Nicholas Johnston

Computing with Management Studies
2001/2002

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)_____

Summary

Digital audio, mainly in the form of MP3 files, is radically altering how we listen to and manage our music. People are increasingly converting their CD collections to MP3 files, allowing them to listen to their music in more flexible ways. Yet this gives rise to a problem. With potentially thousands of songs spread out over numerous disks and CD-ROMs, finding a given song is a frustrating and difficult process.

The aim of this project is to produce a simple application to help a user organise their MP3 collection and to research different technologies (such as XML) for storing music metadata.

An XML DTD, called ‘music list’, has been created to store music metadata. A simple GUI application, Music Organiser, which uses the music list DTD, has been created to demonstrate the viability of using the music list DTD to store music metadata.

Acknowledgements

I would like to thank my project supervisor, Dr. John Stell, for his help, advice, support and encouragement throughout the project process.

I would also like to thank the many others who have helped me with the technical aspects of the project. Special thanks to the members of the ‘PerlMonks’ web site for their invaluable help with any Perl-related problem. Also thanks to the members of the wxPerl mailing list who answered my many questions about wxPerl.

Finally I would like to thank my family and friends for their support during this project and throughout my entire degree.

Contents

1 Introduction **1**

1.1 Background 1

1.2 The Problem 2

1.3 Existing Solutions 3

1.3.1 MPEG Audio Collection 3

1.3.2 Music Library 4

2 Digital Audio and MP3 **5**

2.1 Introduction to Digital Audio 5

2.1.1 Motivation for Compressed Digital Audio 6

2.2 MP3–A Compressed Digital Audio Format 7

2.2.1 Key MP3-related Terms 8

2.2.2 Legal Issues Surrounding MP3 9

2.3 Copyright and Ethical Issues 10

3 XML and Markup **11**

3.1 Introduction to Structural Markup 11

3.2 History of Markup 12

3.3 Introduction to XML 12

4 Design and Implementation **15**

4.1 Design Methodology 15

4.1.1 Extreme Programming 15

4.2 Tools 16

4.2.1 Programming Language 16

4.2.2 GUI Toolkit 18

4.2.3 Data Storage 20

4.3 Project Management 21

4.3.1 Original Schedule 21

4.3.2 Problems 22

4.3.3 Change of Focus 23

4.4 Solution 24

4.4.1 XML DTD 24

4.4.2 Application 27

5 Evaluation and Conclusion **30**

5.1 Evaluation Criteria 30

5.2 User Interface Evaluation 30

5.3 XML DTD Evaluation 33

5.4 Conclusion 35

5.4.1 Future Improvements 37

Bibliography **37**

A Reflection **40**

B Revised Schedule **41**

C User Testing Raw Data **42**

D User Manual **44**

D.1 Introduction 44

D.2 Managing Volumes 44

D.2.1 Adding a Volume 44

D.2.2 Updating a Volume 44

D.2.3 Deleting a Volume 45

D.3 Searching 45

D.4 Browsing and Playing Songs 45

E XML DTD **46**

F Sample XML Document **50**

G Music Organiser Screenshots **53**

Chapter 1

Introduction

1.1 Background

The mass transition from audio cassette to CD marked a significant change in how people organise their music. Music lovers liked the CD's durability—it did not degrade every time it was used, like a cassette. Near-instant access to songs anywhere on the disc and no more 'chewed tapes' made people willing to invest in their music collections.

However, perhaps the most significant change yet in the music world has been the amazing rise in popularity of highly compressed digital audio, primarily in the form of MP3 (MPEG[1] 1 Audio Layer III) files. Critics argue that all MP3s have done is increased piracy and deprived record companies and artists of revenue. This may be partially true, but the mainstream media frequently neglect to mention that certain assumptions[2] made by record companies and intellectual property holders in general about the impact of unauthorised copying are seriously flawed. In addition, the fact that a given technology can be abused should not be considered valid justification for not using it at all.

MP3 and other new and up-and-coming digital audio technologies have had massive impact on the way we listen to and manage our music. These new technologies

[1] Motion Picture Experts Group

[2] It is often said by intellectual property holders that every unauthorised copy made of their property is a loss. Clearly it is *not* a loss unless the person obtaining an illegal copy would otherwise have purchased an original one. This is, of course, not justification for piracy, however.

allow users to enjoy music in ways never before possible, such as:

- Having over 100 average-length songs on a single CD-ROM.
- Creating playlists to play part or all of their collection in a certain order. The uses of this are almost limitless. Users might make a playlist containing all the songs of a particular genre, songs reflecting a certain mood or songs of particular sentimental value to the listener.
- Having thousands of songs easily accessible on their hard disk.

MP3 and new digital audio technologies are not only affecting how music is listened too—it is also affecting how music is created, sold and distributed. Musicians are bypassing the traditional approach to selling music (through a record company) and are now increasingly selling music direct to the public.

MP3 technology, previously only accessible to owners of expensive computers, is becoming far more widespread. The cost of computers in general and CD writers is falling. The use of MPEG (Motion Picture Experts Group) technology in the DVD and MP3 standards means many DVD players can also play MP3 files burnt onto CDR (Compact Disc Recordable), effectively bringing MP3 “into the living room”. Consumer electronics giants such as Panasonic and Goodmans are also launching portable MP3 players that play MP3 files burnt on CD.

1.2 The Problem

All the innovations discussed in section 1.1 go a long way to improving a music lover’s control over his collection, but there is still much to be done. As a user’s collection grows in size, it becomes unfeasible to remember where all his music is located. Finding a particular song or album becomes a tedious process of hunting around numerous disks or CD-ROMs and will often end in frustration. A simple, integrated software package that can organise a user’s music collection in all formats and add value to the music lover’s ‘listening experience’ is required.

A solution is required that

- Conceptually separates a user's collection into *volumes* (a volume might be a directory on fixed media, a removable disk or a network location)
- Stores metadata for each volume in a user's collection
- Permits searching of metadata on a variety of criteria

1.3 Existing Solutions

There are a number of existing programs claiming to help users organise their collections. The existence of other solutions (including several commercial ones) indicates that users *do* want to organise their collections and justifies further research into the subject.

The solution I am proposing, is not, however, just intended to introduce yet another program onto the market. My solution differs significantly in that it uses XML (Extensible Markup Language) (see section 3.3) to store its data. Most existing solutions use proprietary file formats, effectively holding users' data to ransom and committing them to further use of that application. While most solutions do allow data to be exported in some other formats, these are usually only display formats (such as RTF—Rich Text Format) or simple formats such as CSV (Comma Separated Values) that are suitable only for storing tabular data and cannot easily represent complex relationships within data. For example, in CSV, it is difficult to easily represent hierarchical data, yet in XML storing data in a hierarchical manner (by nesting tags) is very common and easy to do.

Using XML to store data on a user's collection is the main differentiating feature of this solution. With the data stored as XML and the DTD (Document Type Definition) readily available, the data is openly stored and will be of the best use to its owner. Using XML also guarantees that the data will be easily usable by other applications for years to come.

1.3.1 MPEG Audio Collection

MPEG Audio Collection (MAC) is a freeware "handy program designed to organize and catalog your MPEG audio file collection" [5]. In addition to just organis-

ing a digital audio collection, MAC provides several other features including:

- A utility to create CD case sleeves.
- A mass file-renaming feature, allowing the user to rename files to a user-defined format.
- An ID3 tag (see section 2.2.1) editor.

MAC uses a similar interface to most music organiser applications. Volumes and their directories are displayed as a tree structure, and when a directory is selected its contents are displayed in a multi-column list box. MAC uses its own file format for data storage.

1.3.2 Music Library

Music Library is a "program [that] can help you manage and organize you music collections such as MP3s, audio CDs or tapes" [18]. As this quotation shows, Music Library takes a different approach to most digital music organisers. Rather than limiting itself to storing details about MP3 files, Music Library stores details on a wide variety of music formats.

Music Library has a vast array of features and would almost be intimidating to a new user. Its interface is similar to MAC, but more complex. It includes a web browser-style "address bar" to browse drives on the user's system, and includes an "alphabet bar" and search bar in the main window.

Music Library's approach to data storage is interesting. It uses the Microsoft Jet database engine to store data, and even allows the user to enter SQL (Structured Query Language) queries to search for items in their collection. Using this approach does, however, add significant overhead to music metadata and makes it almost impossible to access data without using Microsoft tools.

Chapter 2

Digital Audio and MP3

In order to fully understand this project, it is necessary to have an understanding of compressed digital audio in general and of MP3, a digital audio format. This chapter introduces these concepts, the legal issues surrounding them and explains key terms.

2.1 Introduction to Digital Audio

In 1980, Philips Electronics and Sony developed the compact disc audio standard, specified in a document referred to as the "Red Book". The compact disc was the first real digital audio product, marking the start of the digital audio revolution.

To convert audio to digital form, the audio's analog waveform is *sampled*. Sampling is a "process of discretisation in time or space" [4]. In the case of an audio signal, sampling involves taking measurements of *signal amplitude* at regular intervals. The *sampling rate* is the number of times a signal is sampled every second [4]. For CD-quality audio, an analog waveform is sampled at 44.1 KHz (44,100 samples per second). If stereo encoding is used, each sample is recorded once for the left channel of the stereo signal, and once for the right channel. Each sample is stored as a n-bit value, usually 16 bits.

2.1.1 Motivation for Compressed Digital Audio

In section 2.1, the general process of converting audio from analog to digital form was explained. The problem with this approach is that storing audio in digital form requires a vast amount of space. We can calculate the amount of storage required for one second of audio using the following formula:

$$\text{Total size (kilobytes)} = \frac{(\text{Sample rate} \times \text{Bits per sample} \times 2)/8}{1,024}$$

(The multiplication by 2 is included to take account for the left and right stereo channels. The division by 8 is included to convert the bits to bytes, and the division by 1,024 is included to convert from bytes to kilobytes.)

Using this formula, the total size required for one second of stereo audio sampled at 44.1 KHz is:

$$\begin{aligned} \text{Total size per second (kilobytes)} &= \frac{(44100 \times 16 \times 2)/8}{1,024} \\ &= \frac{1411200/8}{1,024} \\ &= \frac{176400}{1,024} \\ &= 172.265625 \end{aligned}$$

A minute of audio would therefore require 10.1 MB of data to store. Assuming an average song length of 3 minutes, a single song would require 30.3 MB of data. While the low cost of storage today might make this seem insignificant, it is not. A large amount of redundant data is being stored and transferring such large files over networks is still relatively tedious. In addition, many PDAs (Personal Digital Assistant) and other mobile computing devices have limited storage capacity, making compression essential.

It should now be clear that to make digital audio more practical, some kind of compression is required. The motivation behind compressed digital audio is simply the sheer size of uncompressed audio.

Lossy and Lossless Compression

Compression algorithms can be placed into two broad categories: lossy and lossless.

In *lossy* compression, some data is discarded during compression and cannot be retrieved during decompression. In other words, if data is compressed and subsequently decompressed, the decompressed data will not necessarily be identical to the original, uncompressed data.

In *lossless* compression, no data is discarded during compression. If data is compressed and subsequently decompressed, the decompressed data will be *identical* to the original, uncompressed data.

Lossy compression is useful for *diffuse* data such as audio and video. Human perception can tolerate minor visual or audio artifacts. For example, we can watch a TV programme with poor reception without significantly reducing the enjoyment of it.

Lossless compression is useful for *symbolic* data such as a spreadsheet. It would clearly not be acceptable to compress a spreadsheet file and discover upon decompression that some values in it had changed!

Most digital audio schemes use *lossy* compression. Using lossy compression usually provides better compression ratios, and by using advanced algorithms, a large amount of data can be discarded without the user noticing.

2.2 MP3–A Compressed Digital Audio Format

In 1988, the Motion Picture Experts Group (a working group of the International Organisation for Standardisation) released the MPEG-1 standard. MPEG-1 Layer 3 specifies how to compress sound, and is commonly referred to as MP3.

MP3 audio uses a *lossy compression* approach to compress audio. Resulting MP3 files are usually about one tenth the size of their uncompressed counterparts. MP3 files are created from 'raw' audio files via an *MP3 encoder*. MP3 encoders use *psychoacoustics* (the study of how people perceive sound) and *perceptual encoding*

to achieve high compression. In these approaches, for example, a note might not be encoded if a louder note is obscuring or blocking it.

Three main factors determine the quality of an MP3 file:

- Bitrate: the bitrate is the number of bits used to represent one second of audio. There is a trade-off between disk space and quality. Usually a bitrate of 128 or 192 kilobits per second (kbps) is used—roughly equivalent to 'CD quality'. A useful analogy for bitrate is drawing a diagram on a piece of paper. If the diagram uses only a small piece of paper, it will appear squashed, untidy and difficult to read. If the diagram uses a larger piece of paper, it will be well spaced out and easy to read, but will occupy more paper. As the size of the diagram increases, readability will increase—but only to a point. Beyond this point, further increases will not actually improve readability but will make handling of the diagram cumbersome.

- Encoder used: there are many MP3 encoders available currently, with significant variation in output quality. Using a poor quality encoder or using an incorrectly configured encoder can result in 'tinny' or 'flat'-sounding output. Currently LAME (a recursive acronym standing for "LAME Ain't an MP3 Encoder") is widely held to be the best MP3 encoder available. LAME provides presets to optimise encoding for different types of music such as rock and classical.

- Original source of audio: MP3 is not a magic wand; it cannot work miracles. If poor quality audio is fed into an MP3 encoder, the output is likely to be equally poor.

2.2.1 Key MP3-related Terms

A number of MP3-related terms are used throughout the remainder of this report. These are explained below.

Encoding Type The above definition of bitrate assumes that CBR (Constant Bitrate) is being used (i.e. the same number of bits are used to encode each second of audio). VBR (Variable Bitrate) encoding can also be used. In this

approach, the number of bits used to encode each second of audio can vary (usually within a user-specified range). The effect of using VBR can be improved quality ('complex' sounds can have the space they require) and usually a reduction in file size ('simple' sounds can use less space).

Stereo Mode MP3 files can be encoded in full stereo, joint stereo or mono. In full stereo mode, data for both stereo channels is stored even if the data in both channels for a given sample is equal. In joint stereo mode, two channels are only stored if their content is different. In other words, "interchannel redundancy [is] exploited" [15, p739]. In mono mode, only one channel is used.

ID3 tag ID3 tags are best thought of as an MP3 file's 'header block'. ID3 tags store metadata such as artist, track title and year of release on their associated track. ID3v2 tags, a new and improved version of ID3, are gradually becoming more widespread. ID3v2 allows for more metadata to be stored, including complex metadata such as time-synchronised lyrics [12].

2.2.2 Legal Issues Surrounding MP3

Although there is a formal ISO (International Organisation for Standardisation) standard for MPEG (and therefore MP3), the MP3 standard is *not* truly open. Fraunhofer IIS-A holds patents on certain key aspects of the MP3 standard. Fraunhofer alleges that no one can create an encoder which *does not* infringe on their patents, even if the encoder is not at all based on the ISO standard reference implementation.

Fraunhofer has waited until MP3 has become widespread before enforcing its patents and demanding royalties, similar to Unisys' tactic when enforcing the patent on the LZW (Lempel, Ziv & Welch) compression algorithm used in GIF (Graphics Interchange Format) files. Fraunhofer has enlisted the services of Thomson Multimedia to collect the fees.

The fees that Fraunhofer are attempting to charge are excessive. The fee to produce an MP3 encoder is $15,000 plus a per-unit fee of $2.50. Fraunhofer are also attempting to charge for *playback devices*, and require anyone using streaming

MP3 audio to pay a fee. These fees are above industry norms and threaten to halt the development of new MP3 consumer electronics devices.

Growing dissatisfaction with these excessive fees has lead to the development of patent-free and open formats such as Ogg Vorbis [3]. Ogg Vorbis claims to be technically superior to MP3 and shows great future potential, however, it is not yet implemented in any consumer hardware devices.

The legal issues surrounding MP3 continue to be a problem—but will not prevent the rise of digital audio as a whole. The benefits of highly compressed digital audio (see section 1.1) are simply too good.

2.3 Copyright and Ethical Issues

> "Piracy is not a technological issue. It's a behavior issue." - Steve Jobs (CEO Apple Computer)

The above quote shows one of the possible views on piracy. Technological solutions are inadequate—time and time again, new encryption and copy protection schemes touted as 'unbreakable' have been cracked only days after release. Technological solutions might slow down piracy, but will certainly not stop it. Technological solutions often constrain legal use of a product and make using the product more difficult. It is for this reason that Steve Jobs chose to make Apple's new 'iPod' portable digital audio player free of any technological solution to piracy [6]. Apple's TV advertising and product packaging have stated "don't steal music" [6].

It is difficult to decide how to tackle the problem of unauthorised copying. Users need to realise when copying is acceptable and when it isn't. Users adopting a more responsible attitude towards copying would generally be a superior solution to ineffective technological schemes.

The copyright and ethical issues relating to digital audio and piracy are complex, and an in-depth discussion is beyond the scope of this project. Indeed, this is a topic worthy of a project in its own right. It will be interesting to see the future approaches taken to solve this problem.

Chapter 3

XML and Markup

3.1 Introduction to Structural Markup

Structured markup "explicitly distinguishes ... the structure and semantic content of a document" [16]. Structured markup does not, in general, store any presentation information—a separate appearance specification (usually a *stylesheet*) can be created and then *applied* to the document [16]. This approach to document creation and data storage has several advantages:

- Semantic markup can make documents "more amenable to interpretation by software" [16]. Versions of a document can be created in other formats and can be tailored to these formats. For example, in a print version of a document, cross references to other parts of the document take the form of the section number. However, an online hypertext version of the document could use hyperlinks to implement cross references.
- The author of a document is freed of presentation and other stylistic issues, allowing them to concentrate on the *content* of the document.
- Data is stored in an open, non-proprietary form[1]. Markup documents become "databases of information. Programs can compile, retrieve, and otherwise manipulate the documents in predictable, useful ways" [16].

[1] It should be noted, however, that some DTDs (Document Type Definition) *are* proprietary, restricting the use of any markup that uses that DTD.

3.2 History of Markup

SGML (Standardised General Markup Language) was the first major markup language. After several years of work by the Computer Languages for the Processing of Text committee of ANSI (American National Standards Institute), SGML was ratified in the ISO 8879 standard [14]. As an interesting aside, the actual published standard itself was written in SGML, and was published in record time after approval [14]!

SGML is widely used. Some of its users include the US Department of Defence, US Internal Revenue Service and the European Community's Office of Official Publications [14]. In addition, many of us use SGML every day without realising it: HTML (Hypertext Markup Language) is an SGML application.

SGML is not without its problems, however. Its main problem is that it has a very flexible and complex grammar, and this makes parsing SGML very difficult and SGML processing software expensive. In addition, most SGML applications use only a tiny subset of the language. SGML's complexity is both a strength and weakness—but it has motivated the development of a more lightweight general purpose markup language, XML [13].

3.3 Introduction to XML

It is difficult to define exactly what XML is. It has been described as a "protocol for containing and managing information" [13, p2] and "a family of technologies that can do everything from formatting documents to filtering data" [13, p2]. XML documents are composed of *tags*. Tags can be nested and can have *attributes* to control optional or additional behaviour of the tag.

A simple XML snippet would be:

```
<directory name="Singles">
 <file name="Elvis Presley - Guitar Man.mp3"
  bitrate="192" encoding="CBR" format="mp3"
  stereo="jointstereo">
   <artist>Elvis Presley</artist>
```

```
   <title>Guitar Man</title>
 </file>
 <cover file="front.jpg" format="JPEG"
  side="front"/>
</directory>
```

This XML snippet shows several of XML's main features:

- `directory`, `file`, `artist`, `title` and `cover` are all examples of *tags*. Tags usually have an *opening* tag, e.g. `<directory>` and an *ending* tag, e.g. `</directory>`.
- `name`, `bitrate`, `encoding`, `format` and `stereo` are all examples of *attributes*.
- The `artist` tag is inside a `file` tag which is in turn inside a `directory` tag. This is called *nesting* and is what makes XML so suited to storing hierarchical data.
- The `cover` tag is an *empty element*—it has no contents. Instead of including a `</cover>` closing tag, a forward slash has been inserted immediately before the tag's closing angle bracket. This is merely a 'syntactic shortcut'.

XML documents can be either *freeform* or *modeled*. Freeform XML is described as "making up your own words but observing the rules of punctuation" [13, p6]. In freeform XML, any tags can appear in any order. Problems arise when tags are misspelt—they will simply be taken to be part of the actual language. For instance, if I had misspelt the `directory` tag in the above example and had instead written `directry`, the XML would still be valid, but would cause problems for a program parsing the XML and expecting to find a `directory` tag.

Providing a *document model* is a far more robust and powerful solution. This is most commonly done with a DTD (Document Type Definition). A DTD is a set of rules or a specification describing what tags can be used in a document and what they can contain [13]. XML documents can be *validated* against a DTD to ensure they are valid. This is a very powerful concept: it makes it possible to check very easily whether or not a document conforms to an exact specification. XML Schema,

an alternative syntax for specifying document models that is currently under development, will make this approach even more powerful. XML Schema allows data types to be specified for attributes. Attributes could be declared as strings, positive integers, dates or even a user-specified pattern. This approach will fundamentally alter data processing since a large proportion of a program's validation code could potentially be carried out automatically by an XML parser.

XML is being used for a wide range of applications:

- SVG (Scalable Vector Graphics) is a vector graphics format defined in XML.
- RDF (Resource Description Framework) is an XML application that provides a framework for web-based metadata.
- MathML (Mathematics Markup Language) is an XML application that can be used to encode equations. This example in particular is a good illustration of the power of XML. One application might use a MathML document to typeset or display the equation, but another might use it to "solve the equation with a series of a values" [13, p5].

Another very interesting component of XML is XSLT (Extensible Style Language for Transformation). With XSLT, documents can be *transformed* from one form into another. The applications of this are nearly limitless. For example, an XML document used to represent a volume in a user's digital music collection could be:

- Transformed into an HTML document for placing on the user's web site.
- Transformed into a playlist so that all the songs in the XML document could be played easily with an MP3 player.
- Transformed into a format for printing such as PostScript or PDF.

XML is a very powerful way of storing data in a structured, hierarchical form. Its wide support, open and non-proprietary nature guarantees its position as an important data storage system for years to come. Readers wishing to find out more about XML should consult the XML specification, [19].

Chapter 4

Design and Implementation

The purpose of this chapter is to explain how the solution was designed and what tools were used to implement it.

4.1 Design Methodology

4.1.1 Extreme Programming

I used part of the extreme programming methodology in creating the solution. Extreme programming is “a deliberate and disciplined approach to software development” [17]. Extreme programming has a number of key features:

- Code should be written as simply as possible—avoid clever generalisation. Simply written code should be easy to extend any way.
- Use consistent style rules (indentation, identifier names, etc.)
- All programming should be done in pairs.
- All code must have unit tests and must *pass* these tests.
- Little formal documentation for code. Code should be well-commented and easy to follow.

Due to the nature of this project, programming cannot be done in pairs, so this aspect of extreme programming cannot be used. However, it is still possible to obtain some of the benefits of extreme programming without pair programming.

4.2 Tools

The solution being proposed is not ‘specialised’ in the sense that a particular programming language or technology stands out immediately as an obvious means of implementation. The relative advantages of various programming languages must be examined and the most suitable language chosen.

The choice of programming language is based on the relative advantages and disadvantages of programming languages that I am familiar with.

As the program will have a GUI (Graphical User Interface), a suitable GUI toolkit must also be chosen.

4.2.1 Programming Language

I am familiar with a number of programming languages: C++, Java, Visual Basic and Perl. As stated above, none of these languages immediately stands out as an obvious candidate for use, so the advantages and disadvantages of each must be considered in turn.

C++

Although C++ is a higher level language than C, it is still relatively low level. While the new ISO standard for C++ addresses this with templates and the STL (Standard Template Library), C++ programming is still relatively difficult. C++ also requires the programmer to allocate and free memory manually, a complex and error-prone process.

The advantages of using C++ would be:

- It is relatively portable and distribution of the final application would be relatively easy.

- A wide range of GUI toolkits is available for C++, including GTK and QT on UNIX and MFC (Microsoft Foundation Classes) on Windows.

Java

If the program was written in Java, the end-user would have to have the JRE (Java Runtime Environment) installed on their system. As this is a fairly large piece of software, distribution would be made more difficult. Performance of Java code is also a concern, and GUI Java programs often have a certain 'sluggish' feel to them.

There are two main GUI toolkits available for Java, the AWT (Abstract Window Toolkit) and Swing. AWT uses native toolkits on the platform it runs on [11]—but Java developers saw this as a problem, fearing that "AWT applications might be subtly incompatible on different platforms" [11, p361]. This motivated the development of Swing, where "components [...] are implemented in Java itself" [11, p361]. It could certainly be argued that implementing components manually is better from a programmer's perspective[1], but using non-native components can create 'alien' applications. This is clearly not good from an HCI (Human Computer Interaction) perspective. Admittedly, Swing can *emulate* the visual appearance of windows and controls on different platforms, but this appearance is still different from a true, native appearance.

The only real advantage of using Java would be that it is significantly more portable than C++ (in the sense that code requires fewer modifications).

Visual Basic (VB)

This 'language' would be a poor choice. VB applications are restricted to the Windows platform, and while VB makes GUI generation trivial, it is a very unpleasant programming environment and language to use.

The only advantages of using VB would be the ease of distribution of the final program and the ease of GUI generation.

[1] As a programmer does not need to be aware of each individual platform's quirks and slightly different behaviour.

Perl

I have considerable experience of Perl and believe that this would be a suitable language for implementation. Perl is a very high-level, loosely-typed language, providing advanced and high-level data structures built in to the language, and very powerful text-processing support [8]. Perl is not traditionally thought of as a language for creating GUI applications, yet there are several GUI toolkits at the Perl programmer's disposal: primarily Perl/TK and the new wxPerl toolkit.

Perl has many advantages. It is highly portable: well-written Perl code is known to run without any modification on UNIX, Windows and Macintosh systems. A large number of extension 'modules' are freely available on the Perl community's CPAN (Comprehensive Perl Archive Network). Perl's built in regular expressions make it extremely powerful for text processing. The Perl language is very semantically dense: a small amount of code can perform what would take many lines of code in other languages. In addition, Perl's garbage-collection approach leaves the programmer free of memory allocation worries.

The one disadvantage of using Perl is that it is a semi-compiled language. Perl code is not 'compiled' in the sense that it produces a native code executable, rather it is compiled and subsequently executed each time a Perl program is run. Therefore the Perl compiler is required, in some form or other, for a user to run a Perl program.

4.2.2 GUI Toolkit

As previously stated, there are several GUI toolkits available to the Perl programmer. There are Windows-only toolkits (such as Win32::GUI), toolkits or bindings to Linux/UNIX desktop environments (such as GNOME and KDE) and finally more general and portable toolkits such as Perl/Tk and wxPerl.

Perl/Tk is the most commonly used GUI toolkit in Perl. It is reasonably well documented and runs on UNIX and Windows platforms. It tries to *emulate* the appearance of a native application, but does not actually *use* the native GUI functions of

the underlying system. In addition, Perl/Tk is a very difficult and relatively low-level toolkit to use.

wxPerl is a relatively new Perl GUI toolkit. It is based on wxWindows, a "C++ framework providing GUI (Graphical User Interface) and other facilities on more than one platform" [7]. wxPerl is a 'wrapper' to the wxWindows library. wxPerl usually uses the native GUI functions of the underlying system, creating applications that are virtually indistinguishable from their native counterparts. wxPerl is highly portable: it is known to run on most platforms where wxWindows runs, i.e. Windows, UNIX (using GTK+), UNIX (using Motif) and Macintosh. wxPerl provides high-level GUI controls such as toolbars, advanced list controls and tree controls. However, wxPerl provides more than just GUI functions—it provides many other useful features for modern application development including:

- A very easy-to-use and high-level printing and print-preview framework. In wxPerl it is possible to print some text using just this short piece of code:

```
use Wx::HTML;
my $page =
  "<html><body>Hello, world!</body></html>";
Wx::HtmlEasyPrinting->new("Printing")->
  PrintText($page);
```

- Clipboard support.
- An easy-to-use online help framework.
- Network support via socket and protocol classes.

wxPerl is still beta code, but it is relatively stable. The wxPerl developers adopt a conservative release policy—it is likely that many would consider wxPerl a stable 1.0 release as it currently stands.

wxPerl (and wxWindows) is also freely available and open-source. No one company controls wxPerl. This is particularly important in GUI development: GUIs seem to move in and out of fashion very quickly—"code can very quickly become obsolete if it addresses the wrong platform or audience. wxWindows helps to insulate the programmer from these winds of change" [7].

I decided to use wxPerl because:

- Its API (Application Programming Interface) and general style of programming seemed easier, cleaner and more intuitive than Perl/Tk.
- It is portable yet produces applications with the native look and feel of their target platform.
- It is very high-level and therefore works well with the Perl philosophy of getting your job done as easily as possible.

4.2.3 Data Storage

Like many computer applications, this program is fundamentally one of data storage, manipulation and retrieval. Data stored on the user's collection must clearly be persistent between different invocations of the program. It follows from this that there must be some means of storing data on disk.

There are a number of possibilities: a proper RDBMS (Relational Database Management System) could be used to store the information. This approach is likely to be unfeasible since installing a large, complex database server would be beyond the average home computer user and would clearly destroy the 'light-weight' nature of the application.

The Perl DBI module could be used to manage storage. The DBI is Perl's abstract database interface, providing a consistent interface to all kinds of storage [2]: true RDBMSs (e.g. Oracle, Microsoft SQL Server, etc.) as well as ordinary text and CSV files. (The advantage of the DBI is that a program can actually use SQL to store and retrieve information in CSV files!)

The final option is that of XML. XML has increased in use and importance in recent years. I feel that it would be a good choice for a number of reasons [13]:

- XML is becoming increasingly widely used and is likely to become the de-facto standard for cross-platform document and data exchange.
- XML is an open standard produced by the World Wide Web Consortium—it is not tied to the fortunes of one particular company. However, XML is not

a standard designed ‘for the sake of it’. It has the support of top companies in the computing industry and is widely used today.

- XML is both machine and human readable. XML parsers now exist for a wide variety of programming languages and environments. In addition, even a human looking at a well-written XML document will be able to understand it.

- XML uses Unicode as its default character set. Unicode greatly simplifies the storage and distribution of text in different languages and scripts. This is especially important as computing is becoming more accessible to users all over the world, many of whom do not speak English. Unicode resolves the previous problems of having various character encoding schemes for the same language (such as Shift-JIS and Euc-JP for Japanese) and the inevitable confusion this causes.

4.3 Project Management

4.3.1 Original Schedule

The original schedule for the project was as follows:

Project Part or Phase	**Date**
Problem understanding	1st October 2001 - 15th October 2002
Research	16th October 2001 - 14 December 2002
(Christmas vacation and exam period)	14 December 2001 - 31 January 2002
Interface prototype	1st February 2002 - 5th February 2002
Initial user interface evaluation	6th February 2002
Design and implementation	6th February 2002 - 27th February 2002
Testing	27th February 2002 - 10th March 2002
User testing (interface evaluation)	11th March 2002
Code and interface improvements	12th March 2002 - 18th March 2002
Follow-up user testing	19th March 2002
Report	23rd March - 23rd April (Easter break)

Several serious problems required significant deviation from this schedule and a re-focusing of the entire project. The project's revised schedule can be found in appendix B.

4.3.2 Problems

I encountered a number of problems. Firstly, creating the XML DTD took far longer than expected. Despite its name, XML is *not* a language as such. It is a *metalanguage* for creating other markup languages. This means that creating an XML DTD is really like inventing a new language. Finding validator software to test the DTD also proved difficult. I had not expected creating the XML DTD to be so complex and time-consuming.

Secondly, the wxPerl toolkit caused major problems. Installing wxPerl was difficult. Unlike most Perl modules which are written in Perl, wxPerl is an *XS* module—a module written in some other language (usually C or C++) with a Perl interface. Installing XS modules is more difficult than ordinary Perl modules. In order to install them, it is necessary to have a copy of the compiler used to compile the Perl implementation in use or it is necessary to find a binary version of the module for the platform in use. The binary distribution I obtained did not work with IndigoPerl, the binary Perl distribution I was using, so I had to switch to ActivePerl. Eventually the module did install correctly, but this problem took several weeks to resolve.

Problems with wxPerl were not just limited to installation, however. wxPerl has no significant documentation of its own—users are merely pointed towards the wxWindows (C++) documentation which contains very minimal notes about some cases where the Perl version differs. Clearly C++ and Perl are very different languages, and attempting to use the C++ documentation was very difficult at first. For example, given a method declaration like:

```
wxTreeItemId AddRoot(const wxString& text, int
  image = -1, int selImage = -1,
  wxTreeItemData* data = NULL)
```

I had to 'convert' this into Perl form, i.e. `AddRoot` is a method of `Wx::TreeCtrl` objects that returns a reference to a `Wx::TreeItemId` object and takes a reference to a `Wx::TreeItemData` object as a parameter, e.g.

```
my $treeId = $tree->AddRoot('Root', -1, -1,
                Wx::TreeItemData->new('Foo'));
```

This process was difficult at first and it significantly slowed down my progress. My main sources of help for learning how to use wxPerl were the sample programs and very basic online tutorials. The sample programs were somewhat of a double-edged sword: while they did show how to use aspects of wxPerl that I needed, they did so in a complicated way (for instance adding menus to alter the behaviour of the example) which made it difficult to understand the example.

4.3.3 Change of Focus

The problems encountered during implementation meant that achieving the project's original aim of a powerful, easy-to-use music organiser application was no longer feasible with the limited amount of time available. I felt it was better to change the focus of the project to examining the viability of using XML to store music metadata. The eventual aim or idea is that this DTD will become an interchange format allowing metadata from one particular music organiser application to be shared with other applications in a seamless and easy manner.

4.4 Solution

4.4.1 XML DTD

The first part of the solution is an XML DTD to specify an XML-based music metadata markup language. The full DTD can be found in Appendix E.

DTD syntax is relatively straightforward. Elements are declared like:

```
<!ELEMENT volume (directory*, file*)>
```

This declares an element called `volume` that contain 0 or more `directory` and `file` elements. DTDs use a similar syntax to regular expressions for denoting the number of elements accepted. For example:

- Including `directory` in an element's declaration means that the element must include *exactly one* `directory` element.
- Including `directory*` in an element's declaration means that zero or more `directory` elements can be included.
- Including `directory?` in an element's declaration means that including `directory` elements is optional.
- Including `directory+` in an element's declaration means that the element must include one or more `directory` elements.

Attributes are declared in a similar fashion:

```
<!ATTLIST cover
  file     CDATA                          #REQUIRED
  format   (JPEG | GIF | PNG)             #IMPLIED
  width    NMTOKEN                        #IMPLIED
  height   NMTOKEN                        #IMPLIED
  side     (front | back | cd | inlay)    #IMPLIED
>
```

This declares attributes for the `cover` element. Each individual attribute's declaration takes the form of the attribute name, its type (character data, an enumeration or name token) and a description of attribute behaviour (i.e. `REQUIRED` means the attribute is required, `IMPLIED` means it is optional).

The DTD's root element is *volume*. This element has a number of attributes:

device The kind of physical device the volume is stored on. This can be fixed media, removable media or a remote location (such as an FTP server).

date The date the volume was created, or, if it has been updated, the date of the last update. Like all dates in the DTD, it is represented in ISO8601 date format.

start The initial directory or "starting point" of the volume. Similar to HTML's `BASE` tag.

serialno The unique serial number of the removable media containing the volume. This can be useful for identifying the media at a later time.

name A user-supplied name of the volume.

`directory` elements can contain unlimited `directory`, `file` and `cover` elements. `directory` elements have just one attribute: *name*, corresponding to the name of the directory.

`file` elements have a number of attributes:

type The type of content stored in the file, i.e. audio or video. For future expansion only, will currently be assumed to be audio.

encoding The type of encoding (CBR or VBR) used. Defaults to CBR.

stereo The stereo mode used.

bitrate The file's bitrate.

format The file's format (MP3, Ogg Vorbis, or WAV).

name The file's name.

size The file's size, in bytes.

`file` elements can also contain other elements:

artist The artist of the track.

title The track title.

duration The track's duration, in seconds.

album The album that the track is taken from.

reldate The release date of the album, in ISO8601 format.

label The track's record label.

genre The track's genre.

misc Miscellaneous data associated with the file.

`directory` elements can include `cover` elements. It is common for users to include scanned image files of a CD's front and back covers in their collection. `cover` elements have a number of attributes:

file The filename of the cover.

format The image file's format. Can be one of JPEG, GIF or PNG.

width The width of the image, in pixels.

height The height of the image, in pixels.

side The side or 'face' of the CD that this is an image of. Can be one of front, back, inlay or CD.

4.4.2 Application

The other part of the solution is a simple application designed to illustrate the feasibility of using XML to store music metadata. This application, Music Organiser, is described in detail in this section.

Music Organiser uses a similar user interface to the program described in section 1.3.1. Volumes and the directories they contain are shown in a tree widget. When a user 'activates' (usually by clicking or pressing enter) a node in the tree widget, the contents of this node are displayed in a multi-column list box opposite. The other functionality of the program, for instance adding a new volume to the collection, is accessed using the pull-down menus. A toolbar is provided to offer quick access to the most commonly-used functions.

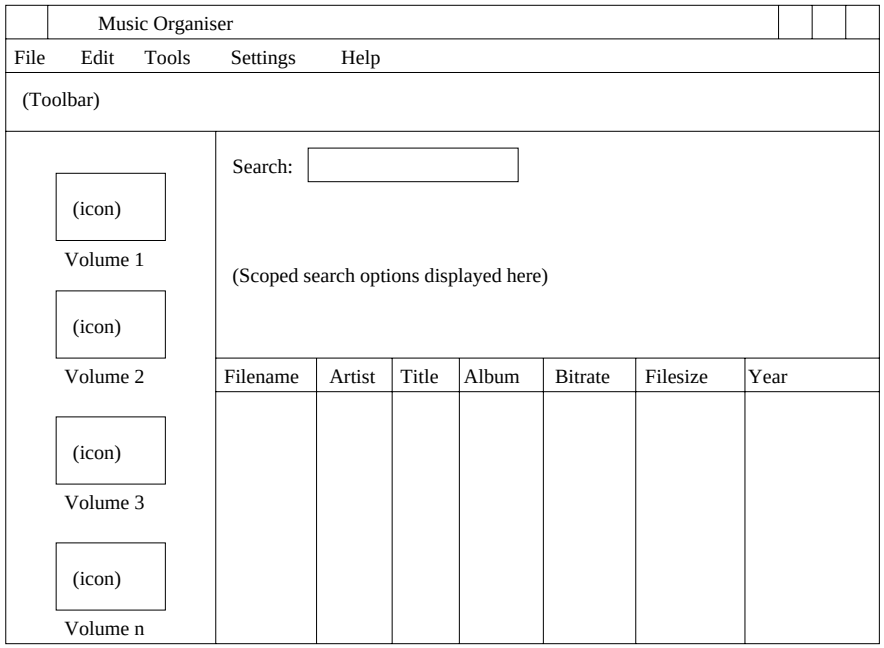

Figure 4.1: Initial User Interface Design

The original user interface design was significantly different to the final interface. The original user interface, illustrated in figure 4.1, did not use a tree structure. Instead, it merely used icons to represent the volumes in a user's collection. The most significant difference, however, is the search feature. Originally I planned to put this in the application's main window. I decided against this since the search

feature would not be used all the time, and should therefore not be visible and occupying space permanently. This follows the idea that "dialogues should not contain information that is irrelevant or *rarely needed*" [9, p20] (emphasis added). A screenshot of Music Organiser's main window, illustrating the final user interface is shown in figure 4.2. Further screenshots illustrating other aspects of the program's interface can be found in Appendix G.

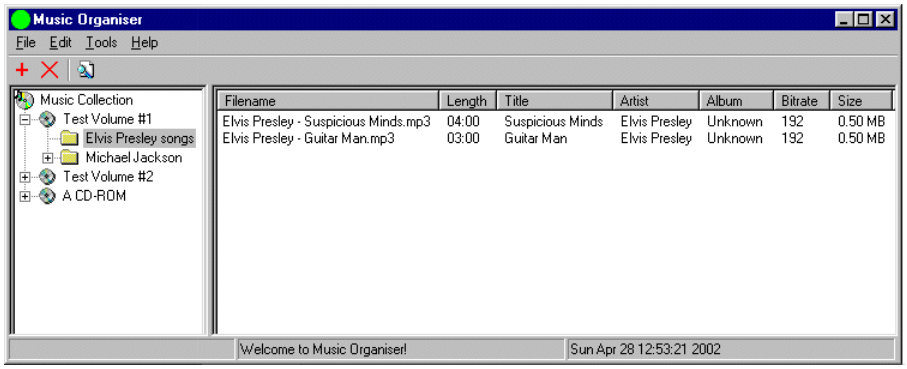
Music Organiser
File Edit Tools Help
Music Collection
Test Volume #1
Elvis Presley songs
Michael Jackson
Test Volume #2
A CD-ROM
Filename Length Title Artist Album Bitrate Size
Elvis Presley - Suspicious Minds.mp3 04:00 Suspicious Minds Elvis Presley Unknown 192 0.50 MB
Elvis Presley - Guitar Man.mp3 03:00 Guitar Man Elvis Presley Unknown 192 0.50 MB
Welcome to Music Organiser!
Sun Apr 28 12:53:21 2002

Figure 4.2: Main Music Organiser Window

Music Organiser is composed of three main components:

- An XML creation subsystem responsible for creating the XML files for volumes in a user's collection. A sample XML document produced by this subsystem can be found in appendix F.
- An XML processing subsystem responsible for interacting with the standard Perl XML parser module, `XML::Parser`. This component parses the XML files representing a user's collection and inserts them into the tree object.
- Code to generate the user interface and event-handling code. Amongst other things, this component is responsible for inserting items into the list box when a tree node is activated. This component is also responsible for most validation—user input is validated before functions in the XML subsystems are called.

The architecture of Music Organiser is illustrated in figure 4.3. I used several existing third-party Perl modules to speed up development of Music Organiser. The

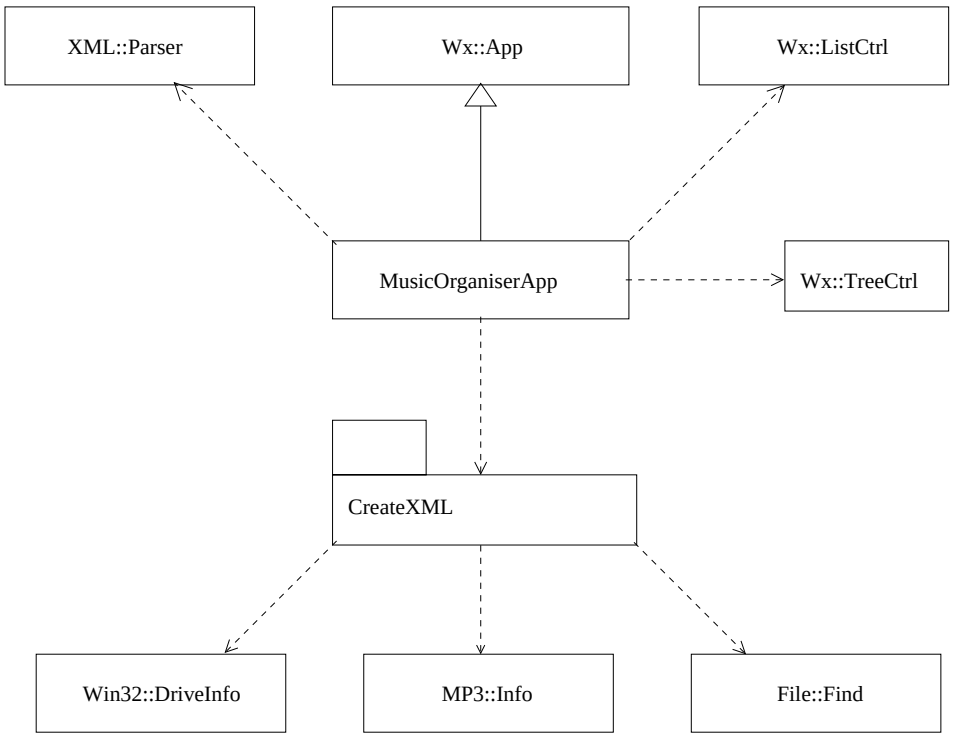

Figure 4.3: Music Organiser Architecture UML Diagram

relationship between these modules and 'native' components of Music Organiser is illustrated in figure 4.3. The purpose and functionality of these modules is briefly described below:

XML::Parser A non-validating XML parser. The parser does, however, check for well-formedness.

Win32::DriveInfo A module that provides information (such as type, free space and serial number) about drives on a Windows system.

MP3::Info A module that provides information (such as artist, album, duration and bitrate) on a given MP3 file.

File::Find A module that uses an event-driven approach to traverse a directory structure.

Chapter 5

Evaluation and Conclusion

5.1 Evaluation Criteria

There are two main parts of the solution that must be evaluated: the user interface and the XML DTD.

The evaluation criteria are:

- The proportion of metadata stored by other music organiser applications that can be represented in the XML DTD.
- The application's ease of use, measured by the number of usability problems revealed in user testing.

5.2 User Interface Evaluation

As previously stated, one of the deliverables for this project is a simple application designed to illustrate the feasability of XML to store music metadata.

I evaluated the user interface of the application using a simple user testing approach. This user testing approach was a combination of

- Task or goal-oriented use. Users were given a number of tasks to carry out:
  - Add a volume

  - Play a song from the volume added
  - Search for a song
  - Delete a volume

- "Thinking aloud" [9]: users were encouraged to say out loud what they were thinking as they were carrying out the above tasks. The value of this approach is that it allows an observer to "determine not just *what* they [the users] are doing with the interface, but also *why* they are doing it" [9, p18].

- Surveying: users were asked to rate the application on a subjective scale for a set of factors based on Jakob Nielsen's *usability heuristics* [9]:

  - The program was easy to use
  - The program used simple and natural language
  - The program's error messages were good
  - The program minimised the user's memory load
  - The program used words and terms consistently, and followed platform (Windows) standards
  - The program provided good feedback and status information

Jakob Nielsen's usability heuristics are ten heuristics or factors by which an interface can be measured. These heuristics are a practical and an almost analytical method for analysing interfaces and improving them.

I did not carry out a large scale user test. Jakob Nielsen states that "elaborate usability tests are a waste of resources" [10] and that "the best results come from testing no more than 5 users and running as many small tests as you can afford" [10]. In addition, I simply do not have the resources to conduct a large-scale user test.

I tested three users. Each user was given a brief introduction to Music Organiser including an explanation of the concept of a volume. I strictly obeyed the "shut-up rule" [9, p204] during the testing: I did not give users any assistance in completing the tasks set, as it would be difficult to provide equal assistance to all users, and providing assistance to some users and not others would bias the results.

The user testing revealed two main usability problems in the application.

The first problem was that of playing songs. When an entry in the list box is double-clicked, the song is played in the user's default MP3 player. However, if the song is stored on removable media, Music Organiser prompts the user to insert the relevant removable media. The volume users added was a CD-ROM, so when they tried to play a song from it, Music Organiser prompted them to insert the CD-ROM, even though the CD-ROM was still in the drive. This confused all the test users to varying degrees. Comments such as "But do we have that disc?" and "Oh, is this an error now?" were common. All users did eventually reason that the CD-ROM that Music Organiser was asking for was already in the drive. Users commented "it [Music Organiser] should check the drive first". I agree with the test users' views on this—Music Organiser should only prompt for a CD-ROM if it is not already inserted.

The second problem was a lack of feedback when adding a volume. As stated in section 4.4.2, creating the XML for a volume is handled by a separate subsystem. This XML creation subsystem is not well-integrated into the main application. Adding a volume can sometimes take up to a minute, and during this time the application becomes unresponsive. Two of the test users asked questions like "Is it doing it now?" and "Is it working?". One user guessed that the program was still functioning correctly by noticing the noise from the CD-ROM drive, but users should not have to rely on such primitive mechanisms for feedback. Again, I agree with the test users' views—Music Organiser should provide some form of progress bar and change the mouse pointer to an hourglass symbol to indicate that the application is busy.

A few other problems were also discovered. One user had difficulty using the search function to find the test song because the search function is case sensitive. Another user thought the menu names and location of menu items were unintuitive. More positively, however, all users correctly guessed that it was necessary to *select* a volume before it could be deleted and all users seemed comfortable with the tree structure and how it related to the list box. The raw results from the user testing can be found in appendix C.

5.3 XML DTD Evaluation

In addition to evaluating the application, the XML DTD was evaluated in the context of its viability as an interchange format for different music organiser applications.

This was done by discovering exactly what metadata the existing solutions described in section 1.3 stored, and then seeing if the XML DTD has provisions for storing all of this metadata in a clean and efficient manner. Discovering the metadata that each program stored for a file was a relatively simple process. In both applications, I viewed the ‘Properties’ or similar window for a given song, and noted the all the different pieces of metadata available. I then considered whether each of these pieces of metadata could be represented using the ‘music list’ XML DTD.

The result of this evaluation is below:

Metadata item	**Music Library**	**MAC**	**Music List XML DTD**
Title	Yes	Yes	Yes
Artist	Yes	Yes	Yes
Album	Yes	Yes	Yes
Genre	Yes	No	Yes
Year	Yes	Yes	Yes
Track number	Yes	Yes	No
Bitrate	Yes	Yes	Yes
Sampling rate	Yes	Yes	No
File size	Yes	Yes	Yes
Duration	Yes	Yes	Yes
Stereo mode	Yes	Yes	Yes
Mood	Yes	No	No
Tempo	Yes	No	No
Occasion	Yes	No	No
Scanned cover image(s)	Yes	No	Yes
Comment	Yes	Yes	No
Record company	No	No	Yes
Release date	No	No	Yes

It is clear from the above table that the XML DTD created in this project is a potentially effective interchange format between music organiser applications. As can be seen in the table above, the following items of metadata cannot be easily stored using the XML DTD:

- Track number: I originally did not include this in the DTD, mainly because I follow the common practice of including the track number in the actual *filename* of the MP3. Not including the track number explicitly in the XML DTD was an error, however—it should not be necessary to parse the track number out of a filename.
- Sampling rate: the vast majority of MP3 files have the same sampling rate, and most users have not heard of this term and are certainly not familiar with what it means. However, this item should be included in the XML DTD

for the sake of completeness and to improve the DTD's ability to act as interchange format.

- Mood, tempo and occasion: only the Music Library application offered these settings. While classifying a song in these ways would be useful, it would be a tedious process and it is difficult to imagine users having the patience to classify each song in their collection in this way. However, it is certainly an interesting idea and it would be very interesting to attempt to devise an automated way of 'guessing' these values.

- Comment: this is an ID3 field containing some comment associated with a file. I originally did not include this in the DTD because this field is rarely used, and when it is used, it usually does not contain any meaningful information. However, for the sake of completeness the XML DTD should be able to represent this data.

It is unsurprising that there is *some* difference in the metadata stored between the two existing solutions described in this report and the XML DTD. The XML DTD was developed in a relatively closed fashion. In an ideal situation, the DTD would be developed through an iterative refinement process based on feedback from authors of music organiser applications. This is emphasised by the fact that two items of metadata, record company and release date, can be represented in the DTD but not in the existing solutions examined. However, even with a feedback or consultation approach it is still likely that some applications will use certain items of metadata that cannot be represented in the XML DTD. The best approach to resolving this situation is to provide a way of storing miscellaneous data, for example via the `misc` element of the music list XML DTD.

5.4 Conclusion

The evaluation criteria set out in section 5.1 have been satisfied to varying degrees.

The proportion of metadata stored by other music organiser applications that can be represented in the XML DTD is relatively high. This was measured by counting the number of items of metadata that were stored in both existing solutions and

seeing how many of these items could be stored in the XML DTD. Both solutions stored 11 identical items of metadata. The XML DTD could represent 9 of these, so a high proportion of metadata can be represented in the XML DTD.

Two significant usability problems were identified during user testing. While these do affect the application's usability, it is good that these problems have come to light and that users generally found the overall interface of the program simple and easy to use.

With these results in mind it can be assumed that on the whole, the aim of the project has been achieved.

5.4.1 Future Improvements

There are several ways in which the solutions provided by this project could be improved.

Music Organiser could be improved in several ways:

- The usability problems detailed in section 5.2 could be fixed.
- Use of *acoustic fingerprinting* along with remote music identification services such as Bitzi [1] could be used to automatically identify music.
- XSLT (see section 3.3) could be used to transform the XML files used to store information about a volume into other formats, such as HTML.

Music Organiser's modular design (see figure 4.3) and separation of interface code from core functionality make it easier to add new functionality to the program. Any new functionality could be implemented as a separate Perl package that can be called from within interface-handling code. Separate Perl packages can still communicate with the interface by being passed a reference to the relevant wxPerl object.

The XML DTD could be improved by adding the 'unsupported' elements (see section 5.3). Two new elements, track number and sampling rate, could be added. These would be permitted only inside `<file>` tags. Finally, the XML DTD could be placed under an open-source license such as the Free Software Foundation's GNU General Public License and placed on a web site, to encourage authors of different music organiser applications to use it.

Bibliography

[1] Bitzi. Bitzi. World Wide Web, 2001. http://bitzi.com/ [12 December 2001].

[2] Alligator Descartes and Tim Bunce. *Programming the Perl DBI*. O'Reilly and Associates, Inc., 2000.

[3] John C. Dvorak. MP3 Gives Way to Ogg Vorbis. World Wide Web, 2000. http://www.forbes.com/2000/09/18/dvorak_index_print.html [31 March 2002].

[4] Nick Efford. *AR21 Handbook*. School of Computing (University of Leeds), 2000.

[5] Jurgen Faul. *MPEG Audio Collection Help*, 2001.

[6] Ian Fried. Apple's iPod spurs mixed reactions. World Wide Web, 2001. http://news.cnet.com/2100-1040-274821.html [5 April 2002].

[7] Julian Smart, Robert Roebling et al. *wxWindows 2.2: A portable C++ and Python GUI toolkit*, 2001.

[8] Larry Wall, Tom Christiansen and Randal L. Schwartz. *Programming Perl*. O'Reilly and Associates, Inc., 2nd edition, 1996.

[9] Jakob Nielsen. *Usability Engineering*. Academic Press, Inc., 1993.

[10] Jakob Nielsen. Test With 5 Users (Alertbox Mar. 2000). World Wide Web, 2000. http://useit.com/alertbox/20000319.html [20 April 2002].

[11] Patrick Niemeyer and Jonathon Knudsen. *Learning Java*. O'Reilly and Associates, Inc., 2000.

[12] Martin Nilsson. ID3v2 Made Easy. World Wide Web, 2000. http://www.id3.org/easy.html [31 March 2002].

[13] Erik T. Ray. *Learning XML*. O'Reilly and Associates, Inc., 2001.

[14] SGML Users' Group. A Brief History of the Development of SGML. World Wide Web, 1990. http://www.sgmlsource.com/history/sgmlhist.htm [3 April 2002].

[15] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, Inc., 3rd edition, 1996.

[16] Norman Walsh and Leonard Muellner. *DocBook: The Definitive Guide*. O'Reilly and Associates, Inc., 1999.

[17] Don Wells. What is Extreme Programming? World Wide Web. http://www.extremeprogramming.org/what.html [1 March 2002].

[18] Wensoftware. *Music Library Help*, 2001.

[19] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. World Wide Web, 2001. http://www.w3.org/TR/2000/REC-xml-2001006 [12 December 2001].

Appendix A

Reflection

This project was challenging and rewarding. It has also been a hugely educational experience. I have learnt more about digital audio, data representation and storage, usability and Perl programming. Although the tools and techniques used in the project are not directly covered by any School of Computing module, the general programming theory and concepts presented in first and second year programming modules were very helpful.

The nature of the project made initial research difficult. Since digital audio is such a fast-moving field, there is little published literature available on the subject. This meant I had to look for information on the Web and attempt to sort the accurate information from the vast amount of outdated, incorrect information available.

I have learnt several valuable lessons from this project. Firstly, I underestimated the amount of work required to become comfortable with the wxPerl GUI toolkit. With hindsight, it is easy to see how I let my confidence in my general Perl programming ability lure me into a false sense of security about my Perl GUI programming skill. Secondly, I underestimated the time required to become comfortable with XML and create an XML DTD. Therefore I would advise anyone considering a similar project or considering using similar tools to become fully comfortable with the tools and techniques they plan to use, well before work on the solution is started. Had I adopted this approach, I might not have encountered the problems that forced a change of the project's focus (see section 4.3.3).

Appendix B

Revised Schedule

As stated in section 4.3.2, a number of problems were encountered that meant the project's original schedule was significantly deviated from.

Milestones in the project's development and completion dates are illustrated in the schedule below.

Milestone	**Completion Date**
XML DTD	17 February 2002
Parsing XML	11 March 2002
XML creation subsystem	10 April 2002
Other application functionality	20 April 2002
User testing	24 April 2002
Write-up	29 April 2002

Appendix C

User Testing Raw Data

This appendix includes the test users' ratings of the application on the criteria listed in section 5.2.

	Strongly Agree				**Strongly Disagree**
The program was easy to use		×			
The program used simple and natural language		×			
The program's error messages were good		×			
The program minimised my memory load		×			
The program used words and terms consistently and followed platform (Windows) standards		×			
The program provided good feedback and status information		×			

	Strongly Agree				**Strongly Disagree**
The program was easy to use		×			
The program used simple and natural language		×			
The program's error messages were good			×		
The program minimised my memory load	×				
The program used words and terms consistently and followed platform (Windows) standards	×				
The program provided good feedback and status information				×	

	Strongly Agree				**Strongly Disagree**
The program was easy to use		×			
The program used simple and natural language	×				
The program's error messages were good		×			
The program minimised my memory load	×				
The program used words and terms consistently and followed platform (Windows) standards	×				
The program provided good feedback and status information		×			

Appendix D

User Manual

D.1 Introduction

Music Organiser is a simple application that helps you organise your MP3 collection. Music Organiser separates your collection into *volumes*. A volume can be either a directory or drive and all its contents. Music Organiser then stores details about these volumes which you can browse and search through.

D.2 Managing Volumes

D.2.1 Adding a Volume

To add a volume to your collection, select ‘Add Volume’ from the ‘Edit’ menu. Select the volume’s starting directory and then enter the volume’s name. In a few moments the newly-added volume will appear in alongside the other volumes in your collection.

D.2.2 Updating a Volume

If you know that a volume has been changed, you can *update* the volume. This simply regenerates the volume, thus taking account of any files that no longer exist or any new files.

To update a volume, first select the volume you want to update. Then select ‘Update Volume’ from the ‘Edit’ menu.

D.2.3 Deleting a Volume

To delete a volume, first select the volume you want to delete. Then select ‘Delete Volume’ from the ‘Edit’ menu.

Music Organiser will ask you to confirm whether you really want to delete the volume or not.

Once you have deleted a volume, you cannot retrieve it.

D.3 Searching

One of the most useful features of Music Organiser is its search feature. To search for a song in your collection, select ‘Search’ from the ‘Tools’ menu. Type in what you want to search for, and select what fields (such as artist, title, etc.) you want to search. Then click ‘Search’ to start the search.

Note that the search feature is case sensitive, i.e. ‘elvis’ would not match ‘Elvis’.

D.4 Browsing and Playing Songs

You can browse your collection by double-clicking on directories within volumes. This will show all the files contained in that directory in the list box.

To play any song, double-click its entry in the list box. You may be prompted to insert a CD-ROM or other removable media if the volume is not stored on fixed media. Songs will be opened in your default MP3 player.

Appendix E

XML DTD

```
<!-- DTD for an XML application to store details of a digital
     audio collection

     February 2002 - Nicholas Johnston.

     Notes:
     - All dates should be represented in the ISO8601 date format,
       where a date like "5 January 2002" would be represented as
       "2002-01-05".
-->

<!-- The root element of the document is a 'volumes' element
     that can have unlimited directory and file elements included
     within. -->
<!ELEMENT volume (directory*, file*)>

<!-- Attributes for the 'volume' element.

     Volumes are constructed in a recursive fashion: the user
     supplies a start directory or location and the list is
     constructed by recursing through this directory or location
     and all its subdirectories.

     Device:
     What kind of physical device the volume is stored on. It
     can be either a CD-ROM, a directory (on fixed media), a
     network location (generally FTP), or a form of removable
     media other than CD-ROM (e.g. Zip disk).

     Date:
     The date on which the volume was created or, if it has
```

```
     been updated, the date of the last update.

     Start:
     The initial directory, drive name or network location where
     recursive volume generation started.

     Serialno:
     An optional attribute for a CD-ROM or other removable media's
     serial number. This can aid identification of different media
     (although these numbers are supposed to be unique, they
     are best thought of as "semi-unique").

     Name:
     A user-supplied name of the volume. This is required since it
     will be user's primary means of identifying volumes
     manually. -->
<!ATTLIST volume
  device   (cdrom | directory | network | removable)   "cdrom"
  date     CDATA                                      #REQUIRED
  start    CDATA                                      #REQUIRED
  serialno CDATA                                      #IMPLIED
  name     CDATA                                      #REQUIRED
>

<!ELEMENT directory (directory*, file*, cover*)>

<!-- Attributes for the directory element:

     name:
     the name of the directory

     absname:
     the absolute name or path to the directory. This is not
     required. Is included since some processing software may find
     it difficult to infer the absolute directory name from the
     XML hierarchy.

     CDATA is used as attribute type since NMTOKENS removes
     whitespace, and users regularly use space in directory
     names. -->
<!ATTLIST directory
  name    CDATA                                        #REQUIRED
  absname CDATA                                        #IMPLIED
>

<!-- A 'file' element is a container element that stores details of
     a given file. -->
<!ELEMENT file (artist, title, duration, album?, reldate?, label?)>
```

```
<!-- Attributes for the 'file' element.

     Type:
     For future expansion: not required and will be assumed to
     be simply "audio".

     Encoding:
     Represents whether Constant Bitrate or Variable Bitrate
     encoding is used. Defaults to CBR since this is by far the
     most common encoding.

     Stereo:
     Certain file formats such as MP3 permit various stereo modes
     such as mono, full stereo, joint-stereo.

     File format:
     The type of file format, such as MP3 or Ogg Vorbis.

     Name:
     The file's filename.

     Size:
     Size of the file in bytes. -->
<!ATTLIST file
  type     CDATA                                    #IMPLIED
  encoding (CBR | VBR)                              "CBR"
  stereo   (mono | stereo | jointstereo)            #REQUIRED
  bitrate  (64 | 128 | 192)                         #REQUIRED
  format   (MP3 | oggvorbis | WAV | PCM)            #REQUIRED
  name     CDATA                                    #REQUIRED
  size     NMTOKENS                                 #REQUIRED
>

<!-- The following elements are allowed only inside 'file' elements.
     The type of PCDATA is used since entity references might be used
     to shorten names of record companies. -->
<!ELEMENT     artist          (#PCDATA)>
<!ELEMENT     title           (#PCDATA)>
<!ELEMENT     duration        (#PCDATA)>  <!-- in seconds -->
<!ELEMENT     album           (#PCDATA)>
<!ELEMENT     reldate         (#PCDATA)>  <!-- release date -->
<!ELEMENT     label           (#PCDATA)>  <!-- record company -->

<!-- miscellaneous info -->
<!ELEMENT     misc            (#PCDATA)>

<!-- genre - one of the defacto "Winamp" genres -->
```

```
<!ELEMENT     genre           (#PCDATA)>

<!-- 'Cover' element is used to store details on any scanned CD
     cover images for a particular directory. -->
<!ELEMENT     cover    EMPTY>

<!-- Attributes of the 'cover' element
     File:
     The image's filename.

     Format:
     One of three possible values representing the image's
     format. Can be either JPEG, GIF or PNG.

     Width:
     Width of the image, in pixels.

     Height:
     Height of the image, in pixels.

     Side:
     Used to represent what 'side' or 'face' of the CD
     this image is of. -->
<!ATTLIST cover
  file     CDATA                                    #REQUIRED
  format   (JPEG | GIF | PNG)                       #IMPLIED
  width    NMTOKEN                                  #IMPLIED
  height   NMTOKEN                                  #IMPLIED
  side     (front | back | cd | inlay)              #IMPLIED
>
```

Appendix F

Sample XML Document

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE volume SYSTEM "musiclist.dtd">
<volume device="fixed" date="2002-04-28" start="C:\mp3\test"
 name="Testing">
<directory name="Edward Shearmur - K-PAX OST"
 absname="C:\mp3\test\Edward Shearmur - K-PAX OST">
<file name="10 - Edward Shearmur - Powell's Return.mp3"
 bitrate="192" encoding="CBR" size="1716059"
 format="mp3" stereo="jointstereo">
 <artist>Edward Shearmur</artist>
 <title>Powell's Return</title>
 <album>K-Pax Soundtrack</album>
 <duration>71</duration>
</file>
<file name="02 - Edward Shearmur - Good Morning Bess.mp3"
 bitrate="192" encoding="CBR" size="4028209"
 format="mp3" stereo="jointstereo">
 <artist>Edward Shearmur</artist>
 <title>Good Morning Bess</title>
 <album>K-Pax Soundtrack</album>
 <duration>167</duration>
</file>
<file name="03 - Edward Shearmur - Taxi Ride.mp3"
 bitrate="192" encoding="CBR" size="5543520"
 format="mp3" stereo="jointstereo">
 <artist>Edward Shearmur</artist>
 <title>Taxi Ride</title>
 <album>K-Pax Soundtrack</album>
 <duration>230</duration>
</file>
<file name="04 - Edward Shearmur - Constellation Lyra.mp3"
```

```
 bitrate="192" encoding="CBR" size="3875236"
 format="mp3" stereo="jointstereo">
 <artist>Edward Shearmur</artist>
 <title>Constellation Lyra</title>
 <album>K-Pax Soundtrack</album>
 <duration>161</duration>
</file>
<file name="05 - Edward Shearmur - Bluebird.mp3"
 bitrate="192" encoding="CBR" size="5576121"
 format="mp3" stereo="jointstereo">
 <artist>Edward Shearmur</artist>
 <title>Bluebird</title>
 <album>K-Pax Soundtrack</album>
 <duration>232</duration>
</file>
<file name="06 - Edward Shearmur - 4th of July.mp3"
 bitrate="192" encoding="CBR" size="6112781"
 format="mp3" stereo="jointstereo">
 <artist>Edward Shearmur</artist>
 <title>4th of July</title>
 <album>K-Pax Soundtrack</album>
 <duration>254</duration>
</file>
<file name="07 - Edward Shearmur - Prot Missing.mp3"
 bitrate="192" encoding="CBR" size="3605026"
 format="mp3" stereo="jointstereo">
 <artist>Edward Shearmur</artist>
 <title>Prot Missing</title>
 <album>K-Pax Soundtrack</album>
 <duration>150</duration>
</file>
<file name="08 - Edward Shearmur - Sarah.mp3"
 bitrate="192" encoding="CBR" size="4402492"
 format="mp3" stereo="jointstereo">
 <artist>Edward Shearmur</artist>
 <title>Sarah</title>
 <album>K-Pax Soundtrack</album>
 <duration>183</duration>
</file>
<file name="09 - Edward Shearmur - New Mexico.mp3"
 bitrate="192" encoding="CBR" size="9229293"
 format="mp3" stereo="jointstereo">
 <artist>Edward Shearmur</artist>
 <title>New Mexico</title>
 <album>K-Pax Soundtrack</album>
 <duration>384</duration>
</file><file name="11 - Edward Shearmur - July 27th.mp3"
 bitrate="192" encoding="CBR" size="6735331"
```

```
 format="mp3" stereo="jointstereo">
 <artist>Edward Shearmur</artist>
 <title>July 27th</title>
 <album>K-Pax Soundtrack</album>
 <duration>280</duration>
</file>
<file name="12 - Edward Shearmur - Coda.mp3"
 bitrate="192" encoding="CBR" size="4823168"
 format="mp3" stereo="jointstereo">
 <artist>Edward Shearmur</artist>
 <title>Coda</title>
 <album>K-Pax Soundtrack</album>
 <duration>200</duration>
</file>
<file name="01 - Edward Shearmur - Grand Central.mp3"
 bitrate="192" encoding="CBR" size="6695834"
 format="mp3" stereo="jointstereo">
 <artist>Edward Shearmur</artist>
 <title>Grand Central</title>
 <album>K-Pax Soundtrack</album>
 <duration>278</duration>
</file>
</directory>
</volume>
```

Appendix G

Music Organiser Screenshots

This appendix includes several screenshots of Music Organiser, showing the application from a user's perspective. The screenshots aim to show the full range of functionality provided by the program. The program's main window is shown in figure G.1.

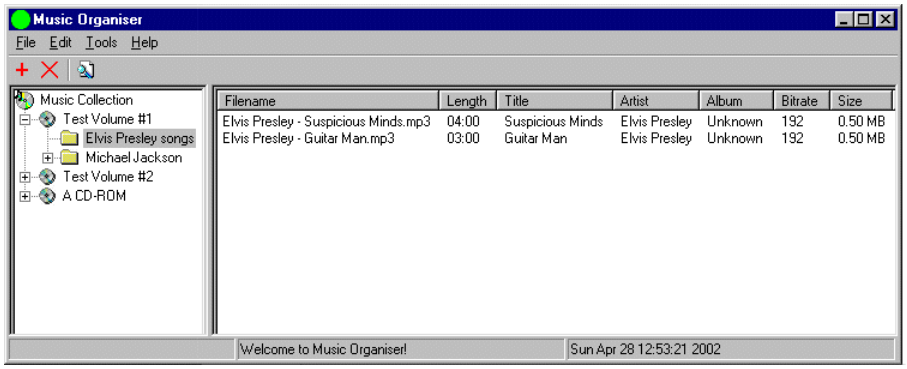
Music Organiser
File Edit Tools Help
Music Collection
Test Volume #1
Elvis Presley songs
Michael Jackson
Test Volume #2
A CD-ROM
Filename Length Title Artist Album Bitrate Size
Elvis Presley - Suspicious Minds.mp3 04:00 Suspicious Minds Elvis Presley Unknown 192 0.50 MB
Elvis Presley - Guitar Man.mp3 03:00 Guitar Man Elvis Presley Unknown 192 0.50 MB
Welcome to Music Organiser!
Sun Apr 28 12:53:21 2002

Figure G.1: Main Music Organiser Window

Adding a volume is a two-step process, as illustrated in figure G.2. The user must first select the starting location of the volume, and then input the volume's name.

Before users can delete a volume, they are prompted with the familiar "are you sure?" question (figure G.3). While some users find this intrusive, it is important to make sure that a volume is not deleted accidentally.

The search feature of Music Organiser is illustrated in figure G.4. Searches are carried out by entering the search string and selecting the scope of the search.

Music Organiser also has considerable validation code. When a user attempts to play a file from a volume stored on removable media, he is prompted to insert the relevant volume (figure G.5). After the user confirms that the removable media is correctly inserted, Music Organiser compares the serial number of the disk in the drive with the serial number stored in the volume's XML file, and issues an error message (figure G.6) if the wrong removable media has been inserted. If a user attempts to play a file from a fixed disk, Music Organiser first checks to see if the file still exists. If the file does not exist, Music Organiser issues an error message, advising the user to update the volume so that files deleted after the original creation of the volume are no longer displayed (figure G.7). If a user attempts to update a volume, Music Organiser checks to see if the start location of the volume exists. If this start location no longer exists, the volume will no longer be accessible and Music Organiser issues an error (figure G.8).

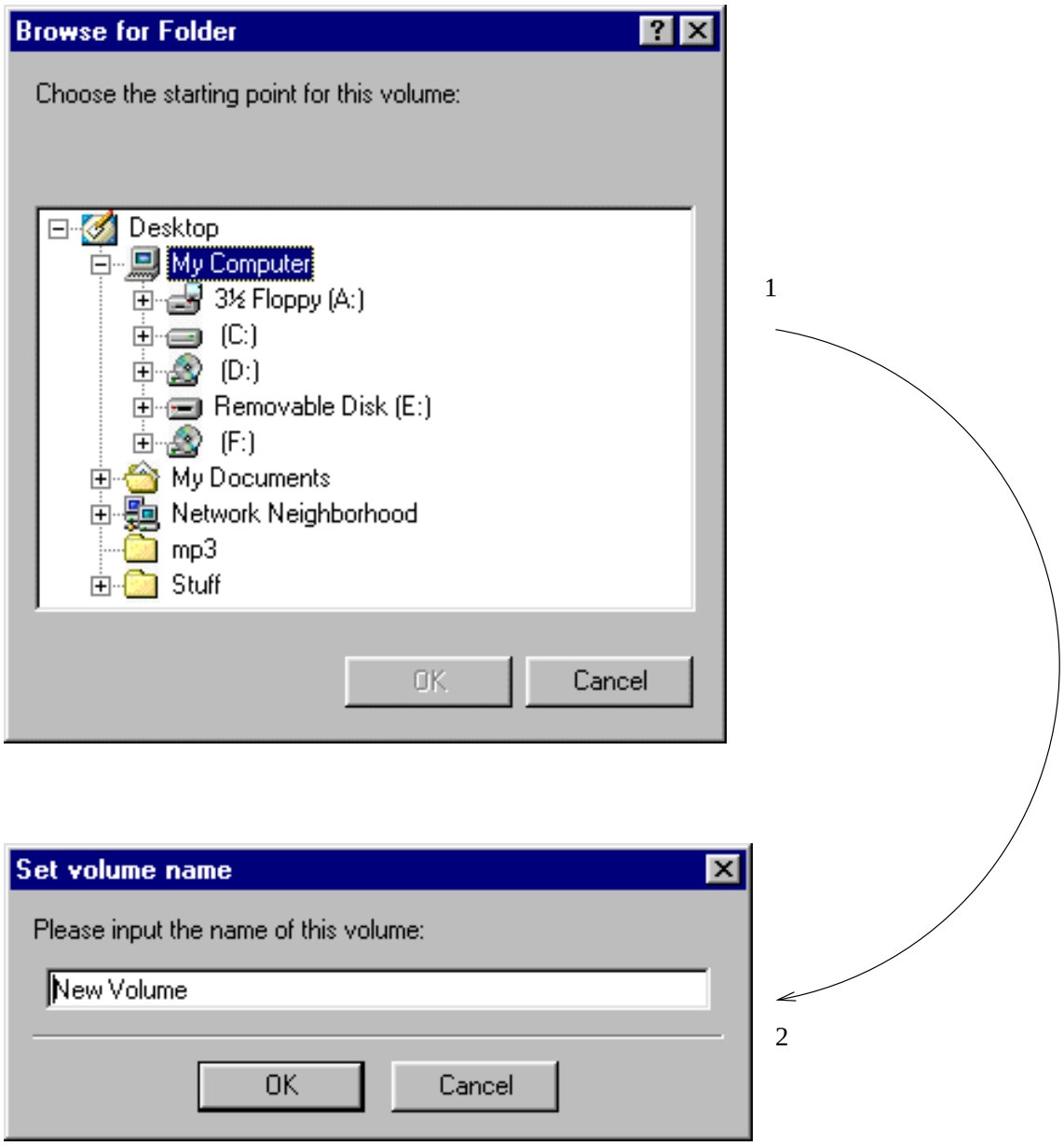

Figure G.2: Adding a Volume

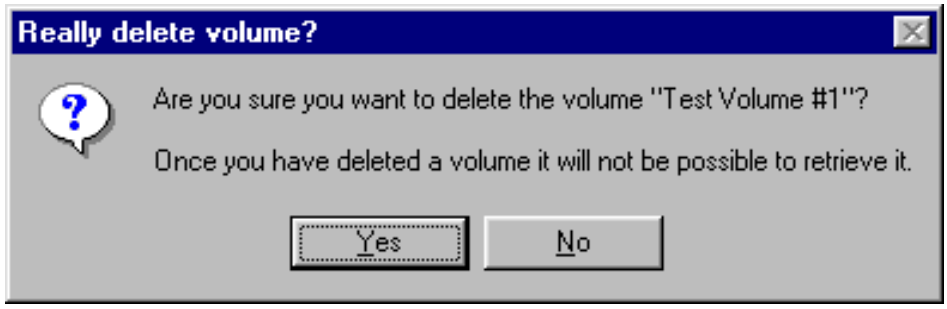
Really delete volume?
Are you sure you want to delete the volume "Test Volume #1"?
Once you have deleted a volume it will not be possible to retrieve it.
Yes
No

Figure G.3: Deleting a Volume

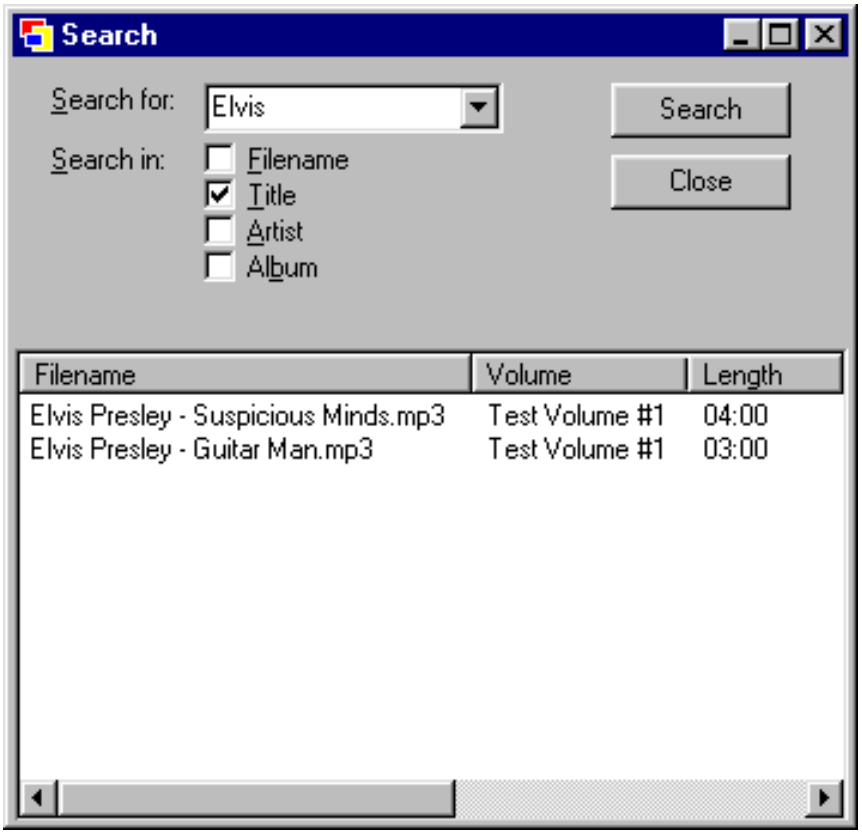

Figure G.4: Searching for Elvis

Insert CD-ROM
To play:
18 - Joy Division - Love Will Tear Us Apart.mp3
you must insert the CD-ROM (with serial number E495:13E4) that holds volume "A CD-ROM" into drive F:.
OK
Cancel

Figure G.5: Prompting to Insert Removable Media

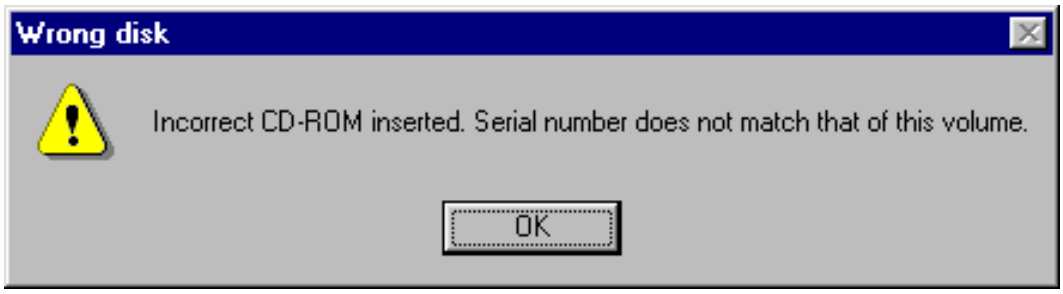

Figure G.6: Wrong Removable Media Inserted

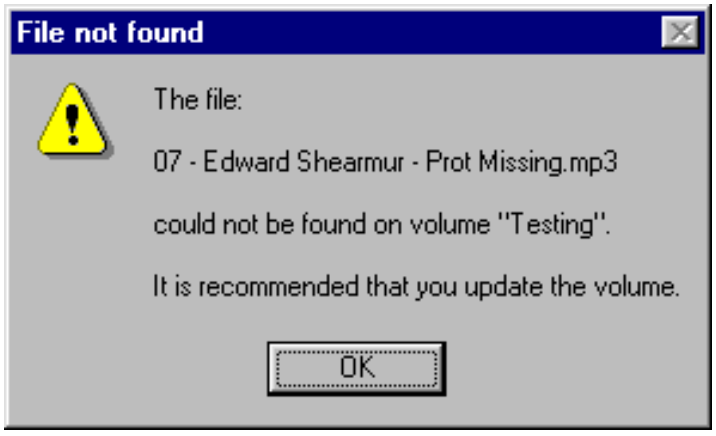
File not found
The file:
07 - Edward Shearmur - Prot Missing.mp3
could not be found on volume "Testing".
It is recommended that you update the volume.
OK

Figure G.7: File Not Found

Cannot update volume
The start location for this volume no longer exists.
It will not be possible to update this volume.
OK

Figure G.8: Starting Location No Longer Exists